集成算法是将其他基础模型进行组合的一中算法。spark.mllib支持两种主要的集成算法：GradientBoostedTrees和RandomForest。 两者都使用决策树作为基础模型。

1.梯度提升树和随机森林

Gradient-Boosted Trees（GBTs）和Random Forest都是用于学习树集成的算法，但训练过程是不同的。 有几个实际的权衡点：

* GBT一次训练一棵树，因此它们比随机森林需要更长时间的训练。 随机森林可以并行训练多棵树。

另一方面，使用具有GBT训练较小（较浅）树比使用随机森林更有优势，并且训练较小树需要的时间更短。

* 随机森林可能不太容易过度拟合。在随机森林中训练更多树可以降低过拟合的可能性，但是使用GBT训练更多树会增加过拟合的可能性。（在统计语言中，随机森林通过使用更多树来减少方差，而GBT通过使用更多树来减少偏差。）
* 随机森林更容易调整，因为性能随树数量增加而改善（对于GBT来说，如果树木数量增长太大，性能可能会开始降低）。

简而言之，两种算法都很有效，并且应基于特定数据集来选择合适的算法。

2 随机森林

随机森林是以决策树作为基础模型的集成算法。随机森林是机器学习模型中用于分类和回归的最成功的模型之一。通过组合大量的决策树来降低过拟合的风险。与决策树一样，随机森林处理分类特征，扩展到多类分类设置，不需要特征缩放，并且能够捕获非线性和特征交互。

spark.mllib支持使用连续和分类特征的二分类和多类分类以及回归的随机森林。spark.mllib使用现有的决策树实现随机森林。

2.1 基本算法

随机森林分别训练一组决策树，因此训练可以并行完成。该算法将随机性注入训练过程，以使每个决策树略有不同。结合每棵树的预测可以减少预测的方差，提高测试数据的性能。

1）训练

注入训练过程的随机性包括：

* 在每次迭代时对原始数据集进行二次采样，以获得不同的训练集（例如，bootstrapping）。
* 考虑在每个树节点处分割的不同随机特征子集。

除了这些随机化之外，决策树训练的方式与单个决策树的方式相同。

2）预测

要对新实例进行预测，随机森林必须整合各个决策树的预测。对于分类和回归，这种整合的方式不同。

分类：多数票原则。 每棵树的预测都算作一个类的投票。预计该标签是获得最多选票的类别。

回归：平均。 每棵树预测一个真实的值。 预测标签是各个树预测的平均值。

2.2 用法提示

我们通过讨论各种参数总结一些使用随机森林的指南。

我们提到的前两个参数是最重要的，调整它们通常可以提高性能：

* numTrees：森林中的树数量。

1)增加树的数量将减少预测的方差，从而提高模型的测试准确性。

2)训练时间大致呈线性增加。

* maxDepth：森林中每棵树的最大深度。

1)增加深度使模型更具表现力和更强大。 然而，深树需要更长的时间进行训练，而且更容易过度拟合。

2）通常，在使用更深的树来训练随机森林比使用单个决策树时更深的树更好。一棵树比随机森林更容易过度拟合（因为森林中多棵树的平均值减少了差异）。

接下来的两个参数通常不需要调整。 但是，他们可以调整以加快训练。

* subsamplingRate：此参数指定用于训练随机森林中每棵树的数据集的大小，作为原始数据集大小的一部分。建议使用默认值（1.0），但减少此分数可以加快训练速度。
* featureSubsetStrategy：用作每个树节点处的分割候选的特征数。该数字被指定为特征总数的占比或函数。减少这个数字会加快训练速度，但如果太低，有时会影响性能。

2.3 代码示例

1）分类

下面的示例演示如何加载LIBSVM数据文件，将其解析为LabeledPoint的RDD，然后使用随机森林执行分类。 计算测试误差以测量算法精度。

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** java.util.HashMap;  **import** java.util.Map;  **import** javax.swing.plaf.basic.BasicInternalFrameTitlePane.SystemMenuBar;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaPairRDD;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.regression.LabeledPoint;  **import** org.apache.spark.mllib.tree.RandomForest;  **import** org.apache.spark.mllib.tree.model.RandomForestModel;  **import** org.apache.spark.mllib.util.MLUtils;  **import** scala.Tuple2;  **public** **class** JavaRandomForestClassificationExample {  **public** **static** **void** main(String[] args) {  SparkConf sparkConf = **new** SparkConf().setAppName("JavaRandomForestClassificationExample").setMaster("local");  JavaSparkContext jsc = **new** JavaSparkContext(sparkConf);  // 加载数据  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(jsc.sc(), path).toJavaRDD();  // 将数据集划分为训练数据和测试数据  JavaRDD<LabeledPoint>[] splits = data.randomSplit(**new** **double**[] { 0.7, 0.3 });  JavaRDD<LabeledPoint> training = splits[0];  JavaRDD<LabeledPoint> testData = splits[1];  // 随机森林模型训练  Integer numClasses = 2;  Map<Integer, Integer> categoricalFeaturesInfo = **new** HashMap<>();  Integer numTrees = 3; // 树的数量  String featureSubsetStrategy = "auto"; // 算法自动选择  String impurity = "gini";  Integer maxDepth = 5;  Integer maxBins = 32;  Integer seed = 12345;  RandomForestModel model = RandomForest.*trainClassifier*(training, numClasses, categoricalFeaturesInfo, numTrees,  featureSubsetStrategy, impurity, maxDepth, maxBins, seed);  JavaPairRDD<Double, Double> predictionAndLabel = testData  .mapToPair(p -> **new** Tuple2<>(model.predict(p.features()), p.label()));  **double** testErr = predictionAndLabel.filter(pl -> !pl.\_1.equals(pl.\_2())).count() / (**double**) testData.count();  System.***out***.println("Test err:" + testErr);  System.***out***.println(model.toDebugString());  }  } |

2）回归

下面的示例演示了如何加载LIBSVM数据文件，将其解析为LabeledPoint的RDD，然后使用随机森林执行回归。 最后计算均方误差（MSE）以评估拟合优度。

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** java.util.HashMap;  **import** java.util.Map;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaPairRDD;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.regression.LabeledPoint;  **import** org.apache.spark.mllib.tree.RandomForest;  **import** org.apache.spark.mllib.tree.model.RandomForestModel;  **import** org.apache.spark.mllib.util.MLUtils;  **import** scala.Tuple2;  **public** **class** JavaRandomForestRegressionExample {  **public** **static** **void** main(String[] args) {  SparkConf sparkConf = **new** SparkConf().setAppName("JavaRandomForestClassificationExample").setMaster("local");  JavaSparkContext jsc = **new** JavaSparkContext(sparkConf);  // 加载数据  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(jsc.sc(), path).toJavaRDD();  // 将数据集划分为训练数据和测试数据  JavaRDD<LabeledPoint>[] splits = data.randomSplit(**new** **double**[] { 0.7, 0.3 });  JavaRDD<LabeledPoint> training = splits[0];  JavaRDD<LabeledPoint> testData = splits[1];  // 随机森林模型训练  Integer numClasses = 2;  Map<Integer, Integer> categoricalFeaturesInfo = **new** HashMap<>();  Integer numTrees = 3; // 树的数量  String featureSubsetStrategy = "auto"; // 算法自动选择  String impurity = "variance";  Integer maxDepth = 4;  Integer maxBins = 32;  Integer seed = 12345;  RandomForestModel model = RandomForest.*trainRegressor*(training, categoricalFeaturesInfo, numTrees,  featureSubsetStrategy, impurity, maxDepth, maxBins, seed);  JavaPairRDD<Double, Double> predictionAndLabel = testData  .mapToPair(p -> **new** Tuple2<>(model.predict(p.features()), p.label()));  **double** testMSE = predictionAndLabel.mapToDouble(pl -> {  **double** diff = pl.\_1() - pl.\_2();  **return** diff \* diff;  }).mean();  System.***out***.println("均方误差:"+testMSE);  }  } |

3.梯度提升树

梯度提升树（GBT）是决策树的集成算法。GBT迭代地训练决策树以最小化损失函数。与决策树一样，GBT处理分类特征，扩展到多类分类设置，不需要特征缩放，并且能够捕获非线性和特征交互。

spark.mllib支持使用连续和分类特征进行二分类和回归的GBT。spark.mllib使用现有的决策树实现来实现GBT。注意：GBT尚不支持多类分类。 对于多类问题，请使用决策树或随机森林。

3.1 基本算法

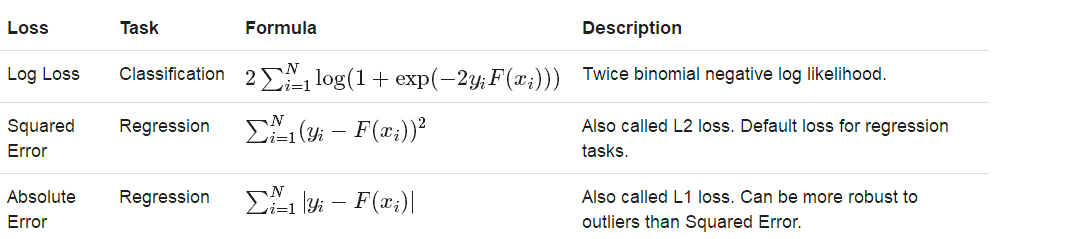
梯度提升迭代地训练一系列决策树。在每次迭代时，算法使用当前集合来预测每个训练实例的标签，然后将预测与真实标签进行比较。重新标记数据集以更加重视预测较差的训练实例。因此，在下一次迭代中，决策树将帮助纠正先前的错误。

重新标记实例的具体机制由损失函数定义（下面讨论）。 每次迭代，GBT进一步减少训练数据的这种损失函数。

* 损失

下表列出了spark.mllib中GBT当前支持的损失。 请注意，每种损失都适用于分类或回归。

注意：N=实例数。 yi =实例i的标签。 xi=实例i的特征。 F（xi）=模型的预测标签。



3.2 用法提示

我们通过讨论各种参数总结了一些使用GBT的指南。

1. loss：根据数据集的不同，不同的损失会产生明显不同的结果。
2. numIterations：设置集成中树的数量。每次迭代都会生成一棵树。 增加此数字可使模型更具表现力，从而提高训练数据的准确性。 但是，如果测试时间过长，则测试时精度可能会受到影响。
3. learningRate：不需要调整此参数。 如果算法行为看起来不稳定，则降低此值可以提高稳定性。
4. algo：使用树[策略]参数设置算法或任务（分类与回归）。

3.3 训练时验证

当训练更多的树时，梯度提升会过拟合。 为了防止过拟合，在训练时进行验证很有用。提供了runWithValidation方法以使用此选项。它需要一对RDD作为参数，第一个是训练数据集，第二个是验证数据集。

当验证误差的改进不超过某个容差（由BoostingStrategy中的validationTol参数提供）时，将停止训练。实际上，验证误差最初会降低，之后会增加。可能存在验证误差不会单调变化的情况，建议用户设置足够大的负容差并使用evaluateEachIteration（每次迭代给出误差或损失）检查验证曲线以调整迭代次数。

1）分类

下面的示例演示了如何加载LIBSVM数据文件，将其解析为LabeledPoint的RDD，然后使用具有日志丢失的Gradient-Boosted树执行分类。 计算测试误差以测量算法精度。

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** java.util.HashMap;  **import** java.util.Map;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaPairRDD;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.regression.LabeledPoint;  **import** org.apache.spark.mllib.tree.GradientBoostedTrees;  **import** org.apache.spark.mllib.tree.configuration.BoostingStrategy;  **import** org.apache.spark.mllib.tree.model.GradientBoostedTreesModel;  **import** org.apache.spark.mllib.util.MLUtils;  **import** scala.Tuple2;  **public** **class** JavaGradientBoostingClassificationExample {  **public** **static** **void** main(String[] args) {  SparkConf sparkConf = **new** SparkConf().setAppName("JavaRandomForestClassificationExample").setMaster("local");  JavaSparkContext jsc = **new** JavaSparkContext(sparkConf);  // 加载数据  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(jsc.sc(), path).toJavaRDD();  // 将数据集划分为训练数据和测试数据  JavaRDD<LabeledPoint>[] splits = data.randomSplit(**new** **double**[] { 0.7, 0.3 });  JavaRDD<LabeledPoint> training = splits[0];  JavaRDD<LabeledPoint> testData = splits[1];  BoostingStrategy boostingStrategy = BoostingStrategy.*defaultParams*("Classification");  boostingStrategy.setNumIterations(3);  boostingStrategy.getTreeStrategy().setNumClasses(2);  boostingStrategy.getTreeStrategy().setMaxDepth(5);  Map<Integer, Integer> categoricalFeaturesInfo = **new** HashMap<>();  boostingStrategy.treeStrategy().setCategoricalFeaturesInfo(categoricalFeaturesInfo);  GradientBoostedTreesModel model = GradientBoostedTrees.*train*(training, boostingStrategy);  JavaPairRDD<Double, Double> predictionAndLabel = testData  .mapToPair(p -> **new** Tuple2<>(model.predict(p.features()), p.label()));  **double** testErr = predictionAndLabel.filter(pl -> !pl.equals(pl.\_2())).count() / (**double**) testData.count();  System.***out***.println("Test Err:" + testErr);  System.***out***.println(model.toDebugString());  }  } |

2）回归

下面的示例演示了如何加载LIBSVM数据文件，将其解析为LabeledPoint的RDD，然后使用带有平方误差的梯度提升作为损失执行回归。 最后计算均方误差（MSE）以评估拟合优度。

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** java.util.HashMap;  **import** java.util.Map;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaPairRDD;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.regression.LabeledPoint;  **import** org.apache.spark.mllib.tree.GradientBoostedTrees;  **import** org.apache.spark.mllib.tree.configuration.BoostingStrategy;  **import** org.apache.spark.mllib.tree.model.GradientBoostedTreesModel;  **import** org.apache.spark.mllib.util.MLUtils;  **import** scala.Tuple2;  **public** **class** JavaGradientBoostingRegressionExample {  **public** **static** **void** main(String[] args) {  SparkConf sparkConf = **new** SparkConf().setAppName("JavaRandomForestClassificationExample").setMaster("local");  JavaSparkContext jsc = **new** JavaSparkContext(sparkConf);  // 加载数据  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(jsc.sc(), path).toJavaRDD();  // 将数据集划分为训练数据和测试数据  JavaRDD<LabeledPoint>[] splits = data.randomSplit(**new** **double**[] { 0.7, 0.3 });  JavaRDD<LabeledPoint> training = splits[0];  JavaRDD<LabeledPoint> testData = splits[1];  BoostingStrategy boostingStrategy = BoostingStrategy.*defaultParams*("Regression");  boostingStrategy.setNumIterations(3);  boostingStrategy.getTreeStrategy().setMaxDepth(5);  Map<Integer, Integer> categoricalFeaturesInfo = **new** HashMap<>();  boostingStrategy.treeStrategy().setCategoricalFeaturesInfo(categoricalFeaturesInfo);  GradientBoostedTreesModel model = GradientBoostedTrees.*train*(training, boostingStrategy);  JavaPairRDD<Double, Double> predictionAndLabel = testData  .mapToPair(p -> **new** Tuple2<>(model.predict(p.features()), p.label()));    **double** testMSE=predictionAndLabel.mapToDouble(pl -> {  **double** diff=pl.\_1()-pl.\_2();  **return** diff\*diff;  }).mean();  System.***out***.println("测试均方误差为："+testMSE);  }  } |